Zentrum für sichere Informationstechnologie – Austria
Secure Information Technology Center – Austria

A-SIT

Seidlgasse 22 / 9, 1030 Wien
Tel.: (+43 1) 503 19 63–0
Fax: (+43 1) 503 19 63–66

Inffeldgasse 16a, 8010 Graz
Tel.: (+43 316) 873-5514
Fax: (+43 316) 873-5520

http://www.a-sit.at
E-Mail: office@a-sit.at

DVR: 1035461
ZVR: 948166612
UID: ATU60778947

SEMANTISCHE SUCHE VERWANDTER CODEMUSTER IN QUELLCODE

Version 1.0 vom 01.09.2020
Johannes Feichtner – johannes.feichtner@a-sit.at

Die Analyse von Quellcode auf sicherheitskritische Elemente kann durch Werkzeuge unterstützt werden, die gewisse, vordefinierte Muster in Programmcode suchen und im Falle von Treffern Aussagen treffen, etwa, dass ein Codefragment mit hoher Wahrscheinlichkeit eine gewisse Funktionalität abbildet oder anfällig zu sein scheint hinsichtlich gewisser Angriffsvektoren. Implizit klammert eine derartige Suche jedoch alle unbekannten Codeteile aus und ist daher inhärent unvollständig.

In diesem Projekt sollen über die Erkennung semantischer Zusammenhänge in Code automatisiert weitere Codefragmente identifiziert werden, die eine ähnliche Funktionalität abbilden. Das Ergebnis könnte dabei helfen, den Zweck unbekannter Codeteile besser einordnen zu können und würde damit die Analyse von Quellcode auf sicherheitskritische Teile maßgeblich unterstützen.

Inhaltsverzeichnis

1. Einleitung

Bei der Entwicklung von Desktop- als auch von Mobilanwendungen ist der Einsatz von Programmbibliotheken, die von Drittherstellern stammen, gang und gäbe. Plattformen wie Github und StackOverflow ermöglichen es EntwicklerInnen, ihre Produktivität durch Verwendung bereits fertiger Code-Schnipsel, die eine gewisse Funktionalität kapseln, deutlich zu steigern. Nicht zuletzt kommt die Vielfalt öffentlich zugänglicher Verzeichnisse mit Quellcode auch der Entwicklung mächtiger Werkzeuge zugute, die etwa auf Anomalie-Erkennung, Code-Vervollständigung, Code-Generierung und Sicherheitsanalyse abzielen.

Um eine gewisse Aufgabe zu realisieren, sind EntwicklerInnen oftmals darauf angewiesen, fremden Code für ihre Zwecke einzusetzen und benötigen daher eine zuverlässige Möglichkeit, zielsicher und effizient dem Zweck entsprechende Codefragmente ausfindig zu machen. Dieser Umstand erklärt die große Popularität hinter den eingangs genannten Plattformen. Oft steht zu Beginn eine Suchanfrage, die folglich auf StackOverflow führt, wo EntwicklerInnen schließlich entdecken, dass ihre Frage bereits zuvor von jemandem gestellt wurde. Sollte eine Antwort mit entsprechendem Codefragment vorliegen, das exakt die eigenen Anforderungen erfüllt, ist die Suche wieder beendet. Ist dies jedoch nicht der Fall, treten die Grenzen aktueller Suchwerkzeuge für Code zum Vorschein: Es ist zurzeit nicht möglich, weitere Exemplare von Codefragmenten zu suchen, die syntaktisch zwar unterschiedlich sind, jedoch semantisch eine äquivalente Funktionalität abbilden.

Analog zur Entwicklung von Programmen spielt dieses Problem unter anderem bei der Sicherheitsanalyse von Mobilanwendungen eine bedeutende Rolle. Ist beispielsweise bekannt, dass in einer gewissen App ein spezifisches Codefragment dafür zuständig ist, Tonaufnahmen über das Mikrofon zu machen und die Aufnahmen im internen Speicher abzulegen, so ist es aktuell nicht möglich, entsprechende Codefragmente auch in weiteren Apps zu finden. Stattdessen beginnt die Analyse der Anwendungen bei jedem Exemplar wieder von vorne. Als bedingte Abhilfe können Werkzeuge eingesetzt werden, die heuristische Vergleiche zwischen Code anstellen oder vordefinierte Muster suchen. Die Krux dabei ist, dass unbekannte Codeteile von der Suche in der Regel ausgeschlossen sind, da eben nur jene gefunden werden können, die schon in irgendeiner Art und Weise bekannt sind. In der Praxis heißt das, dass es entweder statische Regeln gibt, um sie zu identifizieren oder Programmabläufe a priori händisch ein „Label“ zugewiesen bekommen haben. Im Falle von StackOverflow passiert dies etwa durch die aktive Beteiligung von BenutzerInnen, die Fragestellungen semantische Tags hinzufügen können und die die Essenz einer Frage mit wenigen Worten subsumieren. Diese Abstraktion produziert wiederum Metainformation, die dazu verwendet werden kann, Fragestellungen mit analogen Labels bzw. Tags zu finden.

Im Falle von StackOverflow wird das semantische Verständnis von Fragen, Beschreibungen und Code durch eine aktive, von BenutzerInnen bereitgestellte Klassifikation ermöglicht. Die Effektivität von manuellem „Labeling“ bzw. „Tagging“ beruht dabei auf dem ausgeprägten Inhaltsverständnis, sowie der Abstraktionsfähigkeit von Menschen. Intuitiv ließe sich ein vergleichbares Prinzip auch auf Quellcode anwenden, um kontextuelle Informationen zu erfassen und zu abstrahieren: Ein möglicher Weg bestünde etwa darin, einzelne Ausführungspfade in Code zu extrahieren und jeder einen eindeutigen Hashwert zuzuweisen. Typischerweise eingebettet in Ausführungshierarchien könnten auch Merkle-Bäume zum Einsatz kommen, die den Zweck von größeren, zusammenhängenden Codefragmenten identifizieren und aufgrund der Struktur des Baumes auch kleinere subsumieren würden. Damit diese Identifikationsmuster auf jeden Fall eindeutig sind, müsste ein geeigneter Algorithmus dafür sorgen, dass der gelernte Code invariant gegenüber Verschleierung und Compiler-Eigenheiten ist. Die größere Einschränkung ist jedoch die Notwendigkeit einer sog. „ground truth“, also einem Set von Codefragmenten, die manuell „richtig“ kategorisiert wurden. Während monolithische Signaturen also für die Suche nach bestimmten Code-Mustern geeignet erscheinen, etwa um Malware oder Ausführungspfade aufzufinden, die im sicherheitskritischen Kontext problematisch erscheinen, sind diese Ansätze nur in diesem begrenzten Rahmen praxistauglich. Kurzum heißt das, dass sich bestehende Lösungen nicht dafür eignen, beliebige, semantisch verwandte Codemuster zu finden, da eine Suche nur möglich ist, wenn Code im Vorfeld Termini zugewiesen wurden, unter denen sie aufgefunden werden können. Plattformen wie StackOverflow kompensieren dies zu einem gewissen Grad über die Mithilfe von BenutzerInnen. Eine Suche nach Codefragmenten „ohne Label“ ist so aber nicht möglich.

In diesem Projekt verfolgen wir das Ziel, einen Ansatz zu entwickeln, um semantisch verwandte Codefragmente ohne vorangehendes „Labeling“ aufzufinden. Die dabei verfolgte Kernidee ist, dass Quellcode in der Regel genug sprachliche Information beinhaltet, um darüber auch Codefragmente mit verwandter Semantik zu finden. Der Vorteil dieses Zugangs im Vergleich zum herkömmlichen Ansatz liegt vor allem in der Skalierbarkeit d.h. es sollte keine manuelle Intervention mehr vonnöten sein und der zu durchsuchende Korpus ließe sich flexibel und unbeschränkt erweitern. Naturgemäß geht aus einem Codefragment nicht unmittelbar hervor, welche Funktionalität es abbildet. Der nachfolgend präsentierte Ansatz fokussiert sich daher nicht auf die „Intention“ von Code, sondern versucht stattdessen die Semantik von Code in einer Vektorrepräsentation [1] abzubilden. Im Idealfall bedeutet das, dass sich die semantische Ähnlichkeit von zwei Fragmenten auch in der euklidischen Distanz der dazugehörigen Vektoren widerspiegelt.

Der im weiteren vorgeschlagene Verarbeitungsansatz sieht vor, dass für gegebenen Code auf der Ebene einzelner Methoden individuelle Vektoren abgebildet werden. Das Ziel ist es, schlussendlich eine Suchanfrage in natürlicher Sprache ebenfalls im Vektorraum darzustellen und benachbarte Vektoren, die verwandte Codefragmente repräsentieren, als Ergebnis zurückliefern zu können. Die Relevanz der einzelnen Treffer ergibt sich aus der Nähe zum Suchvektor.

2. Quellcode im Vergleich mit normalem Text

Der grundsätzliche Ablauf „semantischer Dokumentensuche“ ist seit einigen Jahren eine zentrale Komponente [2] im Bereich von „Information Retrieval“ in Verbindung mit natural language processing (NLP). Anders als bei regulärem Text, können bei Programmcode jedoch nicht alle Elemente in gleichem Ausmaß herangezogen werden, da beispielsweise Terminalsymbole nicht die gleiche Aussagekraft haben wie die Namen von Variablen. Im weiteren Verlauf wird eine praxistaugliche Lösung erörtert, wie eine semantische Suche verwandter Codemuster in Quellcode möglich ist und welche Eigenschaften in Code dabei besonders relevant sind.

2.1. TF-IDF

Obwohl Quellcode in vielen sprachlichen Aspekten regulärem Text ähnelt, sind die zugrundeliegenden syntaktischen, grammatikalischen und strukturellen Formen unterschiedlich. Etablierte Ansätze sind für die Verarbeitung natürlicher Sprache konzipiert und arbeiten auf der Grundlage von Wortdarstellungen. Neben „wortähnlichen“ Elementen, wie Strings und Identifiers, umfasst die Semantik von Quellcode vor allem Schlüsselwörter, Konstanten, speziell definierte Symbole und mathematische Operatoren. Würden wir einen Operator, wie etwa „=“, einer Wortdarstellung gleichsetzen, würde dieses Wort höchstwahrscheinlich in sehr vielen unterschiedlichen Kontexten vorkommen. Um diese Unausgewogenheit auszugleichen, wurden in den letzten Jahren wissenschaftliche Ansätze entwickelt, die die Relevanz von Wörter nach der Häufigkeit ihres Auftretens gewichten.

Neben der Identifikation semantisch maßgeblicher Elemente in Programmcode, ist auch die Repräsentation für die weitere Verarbeitung eine wesentliche Herausforderung. Um eine semantische Codesuche effizient auch auf große Korpora durchführen zu können, ist der Einsatz von Techniken des maschinellen Lernens nahezu unumgänglich. Entsprechende Ansätze operieren in der Regel jedoch nicht auf der Ebene von Wörtern, sondern erfordern einen numerischen Input. Ein trivialer Ansatz um Wörter numerisch darzustellen wäre etwa die Zuweisung einer Zahl zu jedem Wort, z.B. „Sport“ als Index 1, „Fußball“ als Index 2, etc. Ein Nachteil dieser auch als „One-Hot-Encoding“ bekannten Herangehensweise ist, dass gewisse Algorithmen für maschinelles Lernen dazu tendieren, Ähnlichkeiten zwischen Wörtern daran festzumachen, wie ähnlich der zugewiesene Index ist. Kann jedoch ausgeschlossen werden, dass diese Problematik existiert, so ist „One-Hot-Encoding“ trotz der konzeptionellen Einfachheit des Ansatzes eine praktikable Methode.

Typischerweise wird bei „One-Hot-Encoding“ der numerische Index weiterverarbeitet zu einer binären Sequenz, bei der alle Bits auf 0 gesetzt werden, mit Ausnahme jenes Bits, das den numerischen Index repräsentiert und das auf 1 gesetzt wird. Das Wort „Sport“ mit Index 1 könnte in einem Korpus mit drei Wörtern also repräsentiert werden als „100“. Was aus dieser Darstellung nicht hervorgeht, ist, wie relevant einzelne Wörter in Relation auf gesamte Dokument sind. Alle vorkommenden Wörter gelten somit als gleichermaßen „wichtig“.

In der Praxis haben Konjunktionen wie „und“ oder „oder“ jedoch wenig Aussagekraft und beinhalten vergleichsweise wenig semantische Information. Um einzelnen Termini eine Relevanz zuzuweisen, wurde „Term Frequency – Inverse Document Frequency“ (TF-IDF[1]) entwickelt. Die Grundidee ist, mitzuzählen wie oft ein gewisses Wort in einem Dokument vorkommt und davon abzuleiten, welche Wörter wesentlich dazu beitragen, dass Dokumente als individuell erkannt werden können. Die „Inverse Document Frequency“ stellt die Häufigkeit des Vorkommens einzelner Wörter in Relation zur Häufigkeit ihres Auftretens in einem größeren Korpus von Dokumenten. Einem Wort wird ein hoher TF-IDF-Wert zugewiesen, wenn der Terminus in nur wenigen Dokumenten vorkommt. Wörter, die in einem Dokument (womöglich nur einmal) vorkommen, dies jedoch in allen betrachten Dokumenten geschieht, so erhält es einen niedrigen Wert. Im Kontext semantischer Suche von Quellcode ist diese Eigenschaft anwendbar, um jene Features zu identifizieren, durch die sich ein gewisses Codefragment von anderen unterscheidet.

2.2. Word Embeddings

Ein wesentlicher Nachteil von TF-IDF ist, dass Wörter, die von ihrer Bedeutung her ähnlich sind, nicht notwendigerweise eine vergleichbar hohe TF-IDF-Gewichtung erhalten. Verwenden viele Dokumente beispielsweise das Wort „Auto“, ein weiteres jedoch den Terminus „PKW“, werden beide Wörter als voneinander unabhängig betrachtet. Bei der Suche von semantisch verwandtem Quellcode würde ein derartiger Effekt dazu führen, dass nur Codefragmente mit gleicher Nomenklatur wie ein gegebenes aufgefunden werden könnten.

Um den semantischen Kontext und nicht nur das Vorkommen von Wörtern abzubilden, stellen „Word Embeddings“ einen komplementären Ansatz zu TF-IDF dar. Während „Wort-Vektoren-Abbildungen“ bei TF-IDF über Häufigkeiten des Vorkommens gebildet werden, entstehen „word embeddings“ durch die Repräsentation von Abhängigkeiten über neuronale Netzwerke. Der Kontext in dem ein Wort auftritt bestimmt dabei die numerische Repräsentation. Treten zwei Termini also im gleichen Kontext auf, werden sie durch ähnliche Werte repräsentiert. Ein praktikabler Ansatz zur Erstellung von „Word embeddings“ ist Word2Vec [1] von Mikolov et al. Die dabei verfolgte Idee ist es, „embeddings“ so zu trainieren, dass sie über ihren Kontext definiert werden. Beispielsweise wird im Satz: „Ich habe gestern einen Apfelkuchen gebacken“ das Wort „gebacken“ richtig eingeordnet, indem das neuronale Modell die Kombination „Ich habe gestern einen Apfelkuchen“ lernt und versteht, welches Wort als nächstes kommt.

Eine derartige Darstellung ist auch für die Suche nach semantisch verwandtem Quellcode von zentraler Bedeutung, da Methoden unterschiedlich heißen können. Wenn es etwa gelingt, die in Fragment A vorkommenden Namen so in kontextuellen Vektoren anzuordnen, dass ein Vergleich mit den in Fragment B enthaltenen Namen bzw. Vektoren möglich ist, lässt sich ähnlicher Code auffinden, selbst wenn Variablen oder Methoden im zweiten Fragment anders benannt wurden.

3. Extraktion semantischer Information aus Quellcode

Wir verfolgen die Hypothese, dass Quellcode ausreichend Informationen in natürlicher Sprache enthält, um darauf basierend eine Suche verwandter Codefragmente zu ermöglichen. Dazu erstellen wir auf der Ebene einzelner Methoden und Funktionen „Dokumente“, die in weiterer Folge zu Vektoren transformiert werden können.

Ein einfacher aber pragmatischer Ansatz ist die Verwendung eines sog. „Tokenizers“, um alle Wörter aus Quellcode zu extrahieren und dabei nicht-alphanumerische Zeichen zu ignorieren. Der Vorteil dieser Herangehensweise liegt vor allem darin, dass sie unabhängig von der Programmiersprache ist, da die jeweilige Syntax nicht berücksichtigt wird. Zugleich ist dies jedoch auch ein Nachteil, da so keine Differenzierung mehr möglich ist, zu welchem syntaktischem Konstrukt (Methode, Variable, Schleifenbedingung, etc.) ein gewisser Name gehört. Da „word embeddings“ einzelne Wörter mitsamt ihrem Kontext abbilden, ergibt sich auch daraus implizit eine Zugehörigkeit. Keine Beachtung findet hingegen die in Quellcode abgebildete Baum- bzw. Verzweigungshierarchie. Es sei dahingestellt, inwieweit dies Implikationen auf Suche verwandter Codefragmente hat.

[1] https://de.wikipedia.org/wiki/Tf-idf-Ma%C3%9F

Zur Extraktion einzelner „Tokens“ wird vorgeschlagen, mithilfe eines Parsers einzelne Wörter aus nachfolgenden Elementen einer Methode zu bilden:

- **Methodenname:** Der Name einer Methode wird nach Möglichkeit unterteilt in weitere Segmente. So würde etwa der Name „getPasswordDerivedKey()“ zerlegt in die Wörter „get“, „Password“, „Derived“ und „Key“. Eine entsprechende Erkennung kann natürlich nur heuristisch passieren, z.B. durch Segregation anhand von Großbuchstaben bei einer sog. „Camel-Case-Notation“.

- **Methodenaufrufe:** Zu den wichtigsten Informationsquellen innerhalb eines Codeblocks gehören die Aufrufe weiterer Methoden. Anders als Features wie Abläufe des Kontroll- oder Datenflusses, lassen sich Methodenaufrufe analog zu Methodennamen vergleichsweise einfach in natürlicher Sprache ausdrücken.

- **Enums:** Im Gegensatz zu lokalen Variablen tragen die in Enums verwendeten Werte normalerweise sprechende Namen und beinhalten aussagekräftige Informationen über den Zustand eines Programms.

- **Stringliterale:** Literale beinhalten in der Regel normal lesbaren Text bzw. Code-Dokumentation und damit Wörter, die semantisch von Bedeutung sind.

- **Kommentare:** Wenngleich es keine Formvorschriften für Kommentare in Code gibt, ist die Grundidee, die Funktionalität von Code zu beschreiben. Kommentare sind daher semantisch Paraphrasierungen von Implementierungen bzw. Codemustern.

Wir folgen der allgemeinen Praxis, keine Variablennamen zu extrahieren, da ihre Aussagekraft in der Praxis sehr variiert und unter anderem von Konventionen bei der Entwicklung, sowie der möglichen von Werkzeugen zur Code-Verschleierung negativ beeinflusst werden kann [3].

Nach Extraktion der zuvor angeführten Eigenschaften aus Code, werden die einzelnen Tokens weiterverarbeitet und in einem fiktiven „Dokument“ eingebettet. Zusätzlich zu der zuvor bei Methodennamen exemplarisch erwähnten Aufspaltung von „Camel-Case-Notation“, werden alle nicht-alphanumerischen Zeichen, sowie Wörter mit mehr als 300 Zeichen entfernt. Das Ergebnis ist eine Liste von Wörtern in natürlicher Sprache, die den Quellcode semantisch repräsentieren sollen.

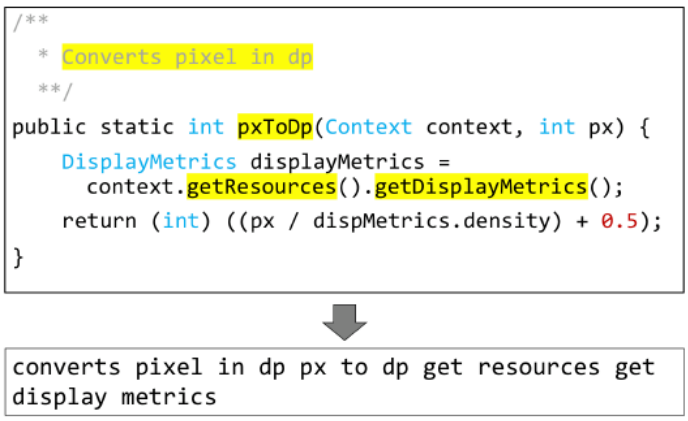

Abbildung 1. Beispiel der aus einer Java-Methode extrahierten Wörter

Abbildung 1 zeigt schematisch das Beispiel der Datenaufbereitung und welche Elemente danach noch übrig bleiben. Die so gewonnenen Wörter können in weiterer Folge in eine hochdimensionale Vektorrepräsentation überführt werden. Wenn die Cosinus-Distanz zwischen einzelnen Vektoren klein ist, haben Wörter verwandte Bedeutungen. Eine Kombination mehrerer Vektoren kann schließlich dazu verwendet werden, um andere Codefragmente mit semantischen ähnlichen Wortvektoren aufzufinden.

4. Fallstudie: Codemuster in Android-Anwendungen

Die einleitend genannten Plattformen StackOverflow und Github bieten bislang keine Möglichkeit, semantisch verwandte Codefragmente aufzufinden. Wie eingangs angemerkt, greift StackOverflow als Abhilfe auf Tags zurück, die von BenutzerInnen einzelnen Postings bzw. Codesnippets zugewiesen werden können. Die Feststellung, was „ähnlich“ ist bzw. in die gleiche Kategorie fällt, bedingt somit eine manuelle Intervention. Bei Github ist eine semantische Suche bisweilen nicht etabliert, wenngleich die Plattform hierzu bereits experimentiert[2]. Abgesehen von EntwicklerInnen ist die semantische Suche verwandter Codemuster auch bei der sicherheitskritischen Analyse von Programmen von wesentlicher Bedeutung. Sind beispielsweise Codefragmente von Malware bekannt und auch, dass die Urheber sie kontinuierlich abändern bzw. weiterentwickeln, kann eine semantische Suche dazu beitragen, auch neuartige Varianten zu finden.

Weitere forschungsrelevante Einsatzszenarien sind im Kontext von Mobilanwendungen zu verorten. Oftmals gibt es mehrere Applikationen, die (vermeintlich) den gleichen Einsatzzweck abdecken. Analog dazu gibt es viele Anwendungen, die ähnliche Codefragmente beinhalten, etwa um Werbung zu integrieren oder auf sicherheitsrelevante Daten von BenutzerInnen zuzugreifen[3]. Semantisch verwandte Codemuster in vielen Apps identifizieren zu können, ist daher von zentraler Bedeutung.

Im Rahmen dieses Projekts wird das in den vorherigen Abschnitten vorgestellte Konzept im Sinne einer Fallstudie mit dem Schwerpunkt auf Codemuster in Android-Anwendungen evaluiert. Die für diese Unternehmung zentralen Fragestellungen sind dabei vor allem:

1. *Lassen sich Apps anhand der verwendeten Codemuster voneinander unterscheiden?*

2. *Implizieren ähnliche Codemuster auch funktional ähnliche Anwendungen?*

Die Beantwortung dieser Fragen erfordert die Verfügbarkeit eines Referenzdatensatzes, der gewonnene Erkenntnisse bestätigen oder widerlegen könnte. In der Praxis ist es jedoch schwierig, Datensätze mit Quellcode zu finden, wo Ähnlichkeiten bzw. Unterschiede bereits festgestellt wurden. Es fehlen also die Vergleichskriterien, um Aussagen zu treffen, inwiefern eine Ähnlichkeit von Codefragmenten vorliegt. Ohne entsprechende Anhaltspunkte kann ein Vergleich aber nur empirisch Aussagen liefern und etwa in gewissen Fällen erfolgreich sein, in anderen gar nicht.

Um das zuvor vorgeschlagene Konzept mithilfe von „word embeddings“ unter realen Bedingungen testen zu können, wird in der Fallstudie auf den Quellcode von Android-Apps zurückgegriffen. Da Anwendungen in der Regel mit einer vom Hersteller verfassten Beschreibung und gewählten Kategorie angeboten werden, bieten sich Vergleichsdaten, die Rückschlüsse darauf erlauben könnten, ob die im Code zweier oder mehrerer Anwendungen gefundenen Ähnlichkeiten auch auf semantische Gemeinsamkeiten im Programmablauf hinweisen.

4.1. Datensatz

Um „word embeddings“ mit den in Quellcode von Android-Apps enthaltenen Wörtern zu trainieren, werden Archive von Android-Apps benötigt. Da ein Crawlen des Google PlayStore technischen Beschränkungen unterworfen ist, verwenden wir für die nachfolgende Analyse die Sammlung des PlayDrone-Projekts [4].

Nach dem Download von rund 2.400 Apps[4] mitsamt Beschreibungen, werden zunächst Cross-Platform-Apps aussortiert, bei denen die Implementierung nicht als Dalvik Bytecode vorliegt, sondern in Form einer JavaScript-Webanwendung realisiert ist. Darüber hinaus wird sichergestellt, dass aus jeder der verfügbaren Kategorien im PlayStore ein gleich großes Sample-Set vorliegt. Dieser „Ausgleich“ über alle Kategorien soll verhindern, dass bereits durch die Auswahl der Apps implizit eine semantische Ähnlichkeit induziert wird, etwa, weil ein Gros der Apps der gleichen Kategorie angehört, und die Ergebnisse der Fallstudie so verzerrt dargestellt werden würden.

[2] https://experiments.github.com/semantic-code-search
[3] https://blog.avast.com/flashlight-apps-on-google-play-request-up-to-77-permissions-avast-finds
[4] https://archive.org/details/playdrone-apks

4.2. Aufbereitung der App-Archive

Für die Extraktion semantischer Information aus Quellcode muss der zu verarbeitende Programmcode zunächst entsprechend aufbereitet werden. Android-Anwendungen liegen grundsätzlich als register-basierter Bytecode der Sprache Dalvik vor. Das Werkzeug baksmali[5] ermöglicht es, Bytecode beliebiger Apps zu disassemblieren und eingesetzte Instruktionen[6] in einer Syntax aufzuschlüsseln, die dem originalen Bytecode entspricht. Die Semantik des Codes wird nicht verändert, der Prozess ist deterministisch und erlaubt somit einen Vergleich von Apps untereinander.

Für die weitere Verarbeitung werden Instruktionen „tokenisiert" d.h. in Wörter unterteilt und sequentiell als Inputdaten übergeben. Nach ersten Experimenten mit den vorliegenden Daten hat sich gezeigt, dass es sinnvoll wäre, Informationen wie Operatoren oder Anführungszeichen gleich im Vorfeld der Verarbeitung aus dem Datensatz zu entfernen. Im Wissen, dass diese Informationen ansonsten ohnehin ausgefiltert werden würden (siehe Abschnitt 2.1), verringert sich zudem das verwendete Vokabular, was wiederum positiv zur Performance beiträgt. Darüber hinaus ist es auch hilfreich, Symbole, Punktationen und alle anderen Zeichen zu entfernen, die weder Zahl noch Buchstabe darstellen. Als letzten Prozess dieses „Feature Engineerings" werden auch Kommentare gelöscht, die im Zuge des Disassemblieren von Android-Apps als „Lesehilfen" eingefügt wurden.

Das Resultat dieser Aufbereitung ist ein sog. „Token Stream", der die Instruktionen des Quelltexts sequentiell wie natürlichen Text abbildet. Wie im vorherigen Abschnitt dieses Dokuments erläutert, ist durch diesen Schritt die hierarchische Struktur verloren gegangen. Durch das (pragmatische) Entfernen von Operatoren und Symbolen, wurde dazu beigetragen, dass TF-IDF diese Informationen nicht von selbst ausfiltern muss und „Word Embeddings" nur aus den verbleibenden Wörtern gebildet werden. Die damit einhergehende Konsequenz ist jedoch auch, dass gewisse Zusammenhänge aus Code entfernt wurden, da der Ausdruck „a = b + 4" nun gleich aussieht wie „a / b & 4". Die in diesem Projekt angestrengte Suche nach verwandten Codemustern wird also nicht in der Lage sein, Ausdrücke anhand der verwendeten Symbole und Operatoren zu unterscheiden.

4.3. Trainieren eines semantischen Modells

Um die Semantik von Code in einer Vektorrepräsentation abzubilden, verwenden wir die „Sequential API" der Keras-Komponente, die in TensorFlow[7] integriert ist. Sie ermöglicht die Aneinanderreihung bzw. sequentielle Ausführung verschiedener Schichten eines neuronalen Netzwerkes und eignet sich, um aus gegebenen Inputdaten zunächst „Word embeddings" herzuleiten und sie anschließend zu lernen. Das Ziel bzw. der „Task", auf den das Modell hintrainiert wird, ist die Vorhersage von einzelnen oder zusammenhängenden Wörtern, die in kontextueller Umgebung einer gewissen Suchanfrage liegen. Die euklidische Distanz der Vektoren dient dabei als Maß für die Ähnlichkeit von Codemustern.

Angelehnt an die von Mikolov et al. [5] vorgeschlagene Architektur eines „Continuous Bag-Of-Words Models" (CBOW) erstellen wir ein einfaches Modell mit mehreren Schichten („Layers"). Eine Alternative zu CBOW wäre der „Skip-Gram"-Ansatz, bei dem auf Basis eines gewissen Zielworts ein Kontext vorhergesagt wird. Da wir gegenständlich jedoch mit Codemustern arbeiten, d.h. nicht nur einzelnen Instruktionen, erscheint CBOW zweckdienlicher. Das vorgeschlagene neuronale Modell umfasst folgende Schichten, die sequentiell nacheinander ausgeführt werden:

1. **„Embedding Layer"**: Die erste Schicht verarbeitet die durch TF-IDF über „One-Hot-Encoding" dargestellte Repräsentation von Wörtern als Input und sucht für den Zeilenvektor eines jeden gegebenen Wort-Index ein passendes „Word Embedding". Diese Vektoren werden adaptiv gelernt, indem die Vektor-Distanzen mit jedem zusätzlichen Trainingsinput („Batch") an die kontextuellen Gegebenheiten angepasst werden. Praktisch heißt das, dass die Darstellung von „Word embeddings" im Vektorraum umso differenzierter wird, je mehr Inputvektoren dem Modell zum Lernen übergeben werden.

[5] https://github.com/JesusFreke/smali
[6] https://source.android.com/devices/tech/dalvik/dalvik-bytecode
[7] https://www.tensorflow.org/guide/keras

Jedes Modell benötigt grundsätzlich auch eine Angabe („Hyperparameter“), wie groß der hochdimensionale Raum sein soll, in dem ein „Word embedding“ repräsentiert werden soll. Empirisch wurde für die in diesem Fall verwendeten Inputdaten festgestellt, dass eine Dimension von 16 eine gute Repräsentation von „Word embeddings“ liefert. Naturgemäß ist dieser Wert nicht generalisierbar, sondern hängt von der „Individualität“ der Inputdaten ab.

2. **„Pooling Layer“:** Die Aufgabe des anschließenden „Pooling Layers“ ist es, Embeddings zu abstrahieren d.h. sog. „Konvolutionen“ durchzuführen, den Durchschnitt von Vektoren zu verwenden und die originalen zu verwerfen. Diese Operation ist notwendig, da nicht alle Wörter bzw. „Word Embeddings“ die gleiche Länge haben. Durch die Berechnung der Durchschnittswerte bzw. „Pooling“ über alle Sequenz-Dimensionen können für alle Inputwerte Vektoren mit konstanter Länge generiert werden.

3. **„Dense Layer“:** Die Vektoren fixer Länge aus der Poolingschicht werden in weiterer Folge über einen „Dense Layer“ mit 16 versteckten Neuronen („hidden neurons“) miteinander verknüpft. Analog zur „Embedding dimension“ wurde auch dieser Hyperparameter-Wert empirisch festgestellt. In dieser Schicht wird also jeder Vektor mit jedem Vektor der vorhergehenden Schicht verbunden. Es ergibt sich somit ein sog. „fully-connected network“.

4. **„Dense Layer“:** In der letzten Schicht wird jeder Vektor mit einem einzelnen Output-Neuron verknüpft. Dieser Schritt ist notwendig, um in der Ausgabe des Netzwerks wieder auf einzelne Vektoren schließen zu können. Über eine sigmoide Aktivierungsfunktion lässt sich somit für jeden Output des Netzwerks ein Wert zwischen 0 und 1 bestimmen, der die Wahrscheinlichkeit (oder Konfidenz) anzeigt, dass ein vorhergesagtes Codemuster eine Ähnlichkeit mit dem als Input gegebenen aufweist.

4.4. Evaluierung

Im Zuge des Trainings eines semantischen Modells mithilfe der im vorigen Abschnitt beschriebenen Architektur, kommt es durch Pooling zu Abstraktionen von Codemustern, was potentiell auch Informationsverlust bedeuten könnte. Um daher fundierte Aussagen über die Qualität des trainierten Modells treffen zu können, ist es essentiell, einen Einblick zu bekommen, wie das finale Modell den in Abschnitt 4.2 beschriebenen Datensatz abbildet.

Bereits nach der ersten Schicht („Embedding layer“) lässt sich ablesen, wie die aus Quellcode extrahierten semantischen Informationen als Vektoren in einem hochdimensionalen Raum dargestellt werden. Die nachfolgende Evaluierung verfolgt in erster Linie das Ziel, die Performance der trainierten Schichten im Hinblick auf die Anwendung mit realen Android-Apps zu überprüfen. Mithilfe von t-SNE [6] und PCA soll stichprobenartig untersucht werden, inwieweit sich auf Basis einzelner Wörter und Kombinationen daraus eine Ähnlichkeit von Apps feststellen lässt.

Die zur Visualisierung zwangsläufig vorzunehmende Dimensionalitätsreduktion kann unter anderem mittels Hauptkomponentenanalyse (PCA) oder t-SNE erreicht werden. Da PCA ein linearer Algorithmus ist, kann er im Gegensatz zu t-SNE nicht immer sinnvoll eingesetzt werden, um komplexe, nicht lineare Abhängigkeiten, wie in diesem Fall vorliegend, abzubilden. Der auf Wahrscheinlichkeitsmodellen basierende t-SNE-Algorithmus ist wiederum auf die Wahl der richtigen „Perplexity“-Stufe angewiesen. Da sich nicht pauschal und vor allem a priori sagen lässt, welcher Algorithmus brauchbarere Visualisierungen liefert, werden in den nachfolgend präsentierten Analysen die Ausgaben von PCA und t-SNE einander gegenübergestellt.

Nach der Extraktion semantischer Informationen aus rund 2.400 Android-Apps, wurden die einzelnen Wörter mittels TF-IDF gewichtet und „Word embeddings“ abgeleitet. Abbildung 2 illustriert die in den 16-dimensionalen Raum projizierten Vektoren nach erfolgter Reduktion der Dimensionalität in einen 2D-Raum. Die Visualisierung über PCA deutet anhand der Gruppierung der Punkte an, dass hinreichende Korrelationen zwischen Wortvektoren hergestellt werden konnten. Praktisch zeigt das, dass die im Datensatz vorgefundenen Wörter öfter vorgekommen sind und es nicht nur dünn besiedelte Matrizen („sparse matrices“) verarbeitet wurden.

Da eine Darstellung über t-SNE oft bessere Einblicke in mögliche Cluster bietet, wurde eine weitere Visualisierung mittels dieses Algorithmus angestellt und PCA gegenübergestellt. Speziell bei t-SNE fällt auf, dass es keine größeren Cluster gibt, sondern sich Wörter in kleineren Strukturen sammeln.

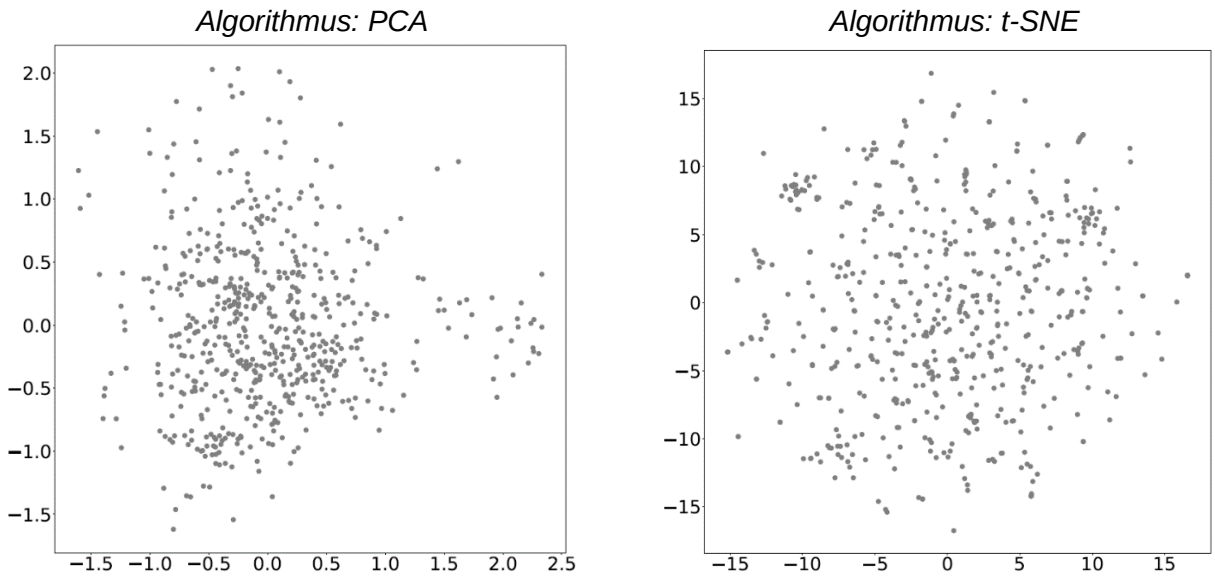

Abbildung 2. Visualisierung von „Word embeddings“ nach Generierung aus Quellcode-Features.

Da sich aus den PCA- und t-SNE-Darstellungen Cluster nicht intuitiv ablesen lassen, ermöglicht die Anwendung von Algorithmen wie k-Means oder DBSCAN einen profunderen Einblick, ob es überhaupt Gruppen von Wörtern bzw. Codemustern geben kann. Für die Anwendung von k-Means ist es notwendig, im Vorfeld bereits festzulegen, wie viele Cluster es geben soll. Naturgemäß ist dieser Wert nicht bekannt und kann nur empirisch herausgefunden werden. Im Gegensatz dazu berechnet DBSCAN die optimale Anzahl an Cluster automatisch. Um neuerlich einen möglichst differenzierten Blick auf „Word embeddings“ zu bekommen, wurden Experimente mit beiden Algorithmen vorgenommen.

Abbildung 3 visualisiert die zuvor generierten „Word embeddings“ mithilfe der Clustering-Algorithmen k-Means und DBSCAN. Farblich hervorgehoben ist erkennbar, dass die semantische Nähe von Wörtern unzweifelhaft abgebildet wurde und sich einzelne Gruppen erkennen lassen. Während k-Means 6 Cluster identifizieren konnte, wurden von DBSCAN 11 Cluster vorgefunden.

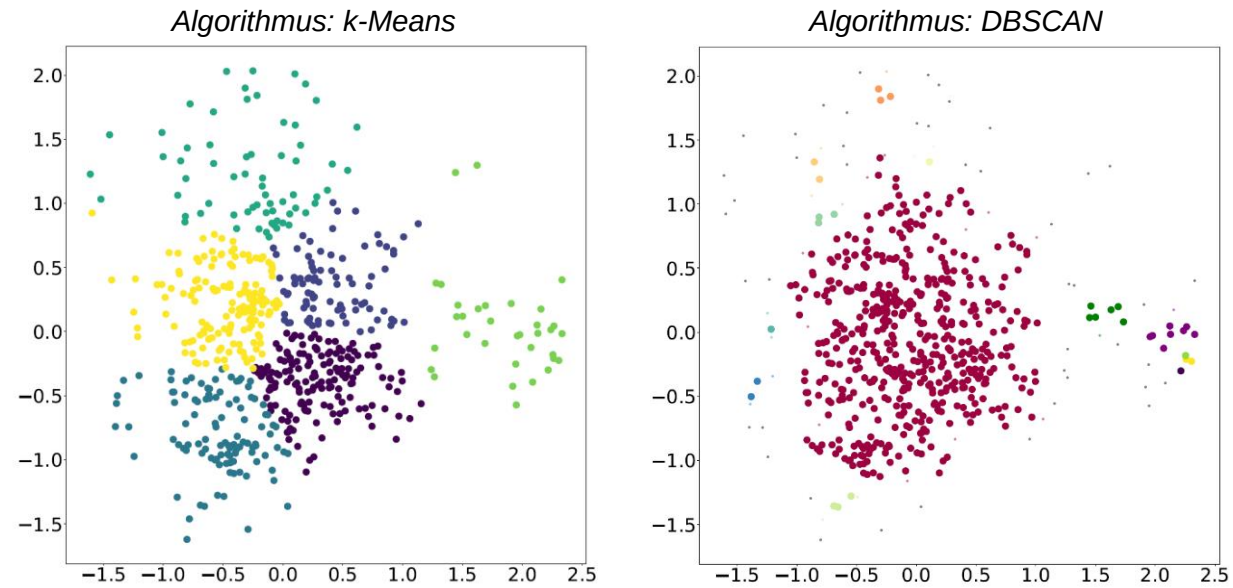

Abbildung 3. Clustering von „Word embeddings“ über k-Means und DBSCAN.

4.5. Fazit der Evaluierung

Aus den vorliegenden Ergebnissen lassen sich folgende Schlussfolgerungen ziehen:

- Die technische Umsetzung und Repräsentation von Code über „Word embeddings“ und die anschließende Abstraktion sowie Verknüpfung einzelner Wörter funktioniert in der Praxis, um semantischen Kontext über ein neuronales Modell abzubilden.

- Die voneinander klar getrennte Anordnung der „Word embeddings“ in Cluster, sowohl bei Visualisierung mittels PCA als auch über t-SNE veranschaulicht, dass es eindeutig Gruppen von Codemustern gibt. Die große Divergenz in den Darstellungen von k-Means und DBSCAN liegt nicht nur in der Wahl der Hyperparameter, sondern zeigt aber auch die Grenzen der verwendeten Algorithmen zur Qualitätsanalyse auf.

- Die euklidische Distanz als Maß zwischen einzelnen Wortvektoren zeigt verlässlich an, inwiefern Wörter in Codemustern semantisch miteinander verwandt sind.

5. Fazit

Im Zuge dieses Projekts wurde der Fokus auf die automatisierte Erkennung semantischer Zusammenhänge in Codefragmenten gelegt. Das Ziel bestand darin, einen Ansatz zu entwickeln, um semantisch verwandte Codefragmente ohne vorangehendes „Labeling“ aufzufinden. Die dabei verfolgte Kernidee war, sprachliche Informationen aus Quellcode zu extrahieren und aktuelle Methoden aus dem Bereich „Information Retrieval“ zur Lösung der gegenständlichen Problemstellung einzusetzen.

Die Vielzahl und der Umfang heutiger Android-Apps verunmöglichen es, alle funktional relevanten Eigenschaften in Apps eindeutig aufzuschlüsseln und zu benennen. Um Zusammenhänge dennoch analysieren zu können, wurde in diesem Projekt ein skalierbarer Ansatz geschaffen, der gezielt Elemente aus Code extrahiert, zu einer Wortrepräsentation aufbereitet und gewichtet. Über die anschließende Verarbeitung in einem neuronalen Netzwerk ist es schließlich möglich, Korrelationen festzustellen und semantisch verwandte Codemuster durch Abstraktion aufzufinden. Die trainierten „Word embeddings“ wurden mithilfe von Algorithmen wie PCA und t-SNE visualisiert. Eine genauere Analyse über k-Means und DBSCAN hat die Existenz von Cluster bestätigt.

Der in diesem Projekt entwickelte Analyseansatz kann eingesetzt werden, um zu einem gegebenen Codemuster semantisch ähnliche Fragmente aufzufinden. Während etablierte Herangehensweisen vor allem auf eine aktive Klassifikation von Code durch Benutzer- bzw. EntwicklerInnen setzen, ermöglicht ein vollautomatischer Weg auch die Identifikation größerer, zusammenhängender Codefragmente. Das Ergebnis unterstützt die sicherheitsorientierte Analyse von Quellcode und trägt insbesondere zu einem besseren Verständnis unbekannter Codeteile bei.

Literaturverzeichnis

[1] T. Mikolov, I. Sutskever und K. Chen, „Distributed Representations of Words and Phrases and their Compositionality," *Neural Information Processing Systems NIPS,* 2013.

[2] X. Ye, H. Shen und X. Ma, „ From Word Embeddings to Document Similarities for Improved Information Retrieval in Software Engineering," *IEEE/ACM 38th International Conference on Software Engineering (ICSE),* 2016.

[3] M. Vijayaraghavan, C. Swarat und C. Jermaine, „Bayesian Sketch Learning for Program Synthesis," *CoRR abs/1703.05698,* 2017.

[4] N. Viennot, E. Garcia und J. Nieh, „A Measurement Study of Google Play," *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems,* 2014.

[5] T. Mikolov, K. Chen, G. Corrado und J. Dean, „Efficient Estimation of Word Representations in Vector Space," *arXiv: https://arxiv.org/pdf/1301.3781.pdf,* 2013.

[6] L. van der Maaten, „Visualizing Data using t-SNE," *Journal of Machine Learning Research, Volume 9,* 2008.

[7] S. Mingshen, M. Li und J. C. Lui, „DroidEagle: Seamless Detection of Visually Similar Android Apps," *8th ACM Conference on Security and Privacy in Wireless and Mobile Networks, WiSec '15.,* 2015.

[8] Y. Shao und X. Luo, „Towards a scalable resource-driven approach for detecting repackaged Android applications," *Proceeding ACSAC '14 Proceedings of the 30th Annual Computer Security Applications Conference,* 2014.

[9] A. Peruma, J. Palmerino und D. Krutz, „Investigating user perception and comprehension of Android permission models," *MOBILESoft '18: Proceedings of the 5th International Conference on Mobile Software Engineering and Systems,* 2018.

[10] S. Ruberto und G. L. Scoccia, „An Investigation into Android Run-time Permissions from the End Users' Perspective," *MOBILESoft '18: Proceedings of the 5th International Conference on Mobile Software Engineering and Systems,* 2018.

[11] S. Lundberg und S.-I. Lee, „A Unified Approach to Interpreting Model Predictions," *Advances in Neural Information Processing Systems 30,* 2017.